Das Generierungswerkzeug unter die Lupe genommen

From Zero to Hero?

Das Aufsetzen neuer Projekte oder das Erweitern bestehender Projekte um neue Technologien ist häufig mühselig und zeitintensiv. Mit Seam Forge betritt ein neues Generierungswerkzeug die Bühne, mit dem das Aufsetzen und die Erweiterung von Projekten automatisiert werden sollen. In diesem Artikel stellen wir das Generierungswerkzeug Seam Forge sowie dessen Konzepte anhand eines Java-EE-6-Beispiels vor.

von Michael Schütz und Marek Iwaszkiewicz

Wer kennt das nicht: ein neues Java-EE-Projekt mit spannender Fachlichkeit und interessanten Technologien soll umgesetzt werden. Doch bevor mit der eigentlichen Umsetzung der fachlichen Anforderungen losgelegt werden kann, muss die Architektur aufgesetzt und alle beteiligten Fragmente und Technologien in Einklang gebracht werden. Dieser Schritt erfordert nicht nur ein umfangreiches technisches Wissen, sondern verschlingt häufig auch einen nicht unerheblichen Anteil des Projektbudgets, von den strapazierten Nerven der beteiligten Entwickler oder Architekten ganz abgesehen. Seam Forge adressiert dieses Problem. Es stellt ein Generierungswerkzeug dar, mit dem ein komplettes und sofort lauffähiges Java-EE-Projektgrundgerüst innerhalb weniger Minuten erstellt werden kann. Jedoch beschränkt sich Forge nicht nur auf das Aufsetzen neuer Projekte, sondern bietet auch die Möglichkeit, bereits bestehende Projekte inkrementell um neue Technologien und Funktionen zu erweitern. Wir stellen Forge zunächst anhand eines einfachen Beispiels vor, in dem ein neues Projekt erstellt wird. Anschließend widmen wir uns den weitergehenden Konzepten und demonstrieren anhand eines Plug-ins die Erweiterbarkeit des Werkzeugs.

Einordnung von Forge

Forge verfolgt einen universellen, technologie-agnostischen Ansatz und will sich nicht nur auf Java-EE-Projekte beschränken. Zwar basiert Forge selbst auf Java, durch ein abstraktes Programmier- und Metamodell soll jedoch die Möglichkeit geschaffen werden, das Werkzeug beliebig erweitern zu können. Dadurch soll die Generierung auch für Technologien und Frameworks einsetzbar sein, die nicht auf Java basieren. Die Grundlage für die Entwicklung des Kerns von Seam Forge bildet jedoch der Java-EE-6-Technologie-Stack. In diesem Sinne sollte die Verbindung zu dem Webframework Seam auch nicht zu strikt gesehen werden. Auf den ersten Blick lässt der Name ausschließlich auf ein Werkzeug für Seam schließen. Mit der Version 3 hat Seam jedoch einen größeren Strukturwandel vollzogen und stellt eine Sammlung aufeinander abgestimmter Java-EE-6-Entwicklungsmodule dar. Seam Forge entspricht einem solchen Modul.

Das Arbeiten erfolgt ähnlich wie bei den Generatoren von Grails und Spring Roo auf der Kommandozeile. Hierfür wird mit der Forge Shell eine eigene Shell zur Verfügung gestellt. Damit hat der Entwickler insbesondere durch die kontextsensitive Autovervollständigung ein komfortables und mächtiges Mittel zur Hand. Das Aufsetzen und konfigurieren der Persistenzschicht ist zum Beispiel in weniger als einer Minute erledigt. Seam Forge liegt derzeit in Version 1.0.0. Beta 1 vor und bietet in der Basiskonfiguration die folgenden Features:

- Erstellung von Java-Webanwendungen „from scratch“
- Konfiguration von Java Persistence (JPA) für Hibernate, EclipseLink und OpenJPA auf JBoss AS 6, JBoss AS 7 und GlassFish
- Konfiguration von Contexts and Dependency Injection for Java (CDI)
- Erstellung und Änderung von Entitäten
- Generierung des Backends und Frontends für CRUD auf Basis einer Entität
- Verwaltung von Projektabhängigkeiten und Repositories

Jumpstart Java EE 6

Wir beschränken uns hier auf die von Forge bereitgestellte Basisfunktionalität, mit der wir in wenigen Schritten ein auf Maven basierendes Projekt mit einer Datenbankanbindung und einem JSF-Frontend erstellen. Dieses Beispiel tangiert bereits einen Großteil der gerade aufgezählten Basisfeatures. Bevor wir mit dem Beispiel loslegen können, steht zunächst die Konfiguration der Umgebung an. Hierzu sind die folgenden Schritte notwendig.

Installation und Vorbereitung

- Seam Forge herunterladen und entpacken
- Umgebungsvariable *FORGE_HOME* setzen und *bin*-Verzeichnis in Path aufnehmen
- Applikationsserver JBoss in Version 6 oder 7 herunterladen

Projekt erstellen

Nachdem die Installation durchgeführt wurde, kann Forge über die Konsole mittels *forge* gestartet werden. Die Erstellung eines Projekts erfolgt mit dem Befehlsaufruf *$ new-project --named testprojekt --topLevelPackage com.acme*. Damit wird zunächst nur das Projekt erstellt. Da die generierten Projekte standardmäßig auf Maven basieren, wird hier der Projektordner mit den bei Maven üblichen Verzeichnissen sowie der POM erstellt. Etwas interessanter wird es im nächsten Schritt, in dem das Scaffolding zum ersten Mal zum Einsatz kommt: *$ scaffold setup*. Der *scaffold*-Befehl bewirkt mit dem Parameter *setup* die Generierung des Grundgerüsts der Webanwendung. Der im Zuge der Befehlsausführung gestellten Frage, ob im Projekt der Maven Packaging Type auf WAR (Webarchiv) geändert werden soll, sollte zugestimmt werden. Die Ausführung des Befehls führt zur Installation der für ein Webprojekt erforderlichen Facets wie dem *WebResourceFacet* und dem *JSFFacet*. Weiterhin sind Angaben zur technischen Konfiguration erforderlich. Zunächst wird die Zielversion des Frameworks Metawidget erwartet. Metawidget [1] ist ein Object/User Interface Mapper (OIM), mit dem CRUD-Formulare dynamisch aus Domainobjekten generiert werden können. Metawidget unterstützt verschiedene Frontend-Technologien. Im Fall von Seam Forge ist das derzeit ausschließlich JSF. Nach der Versionsauswahl wird die entsprechende Abhängigkeit in die POM eingetragen. Nach der Befehlsausführung wurden die in einem Java-Webprojekt erwarteten Verzeichnisstrukturen mit den üblichen Konfigurationsdateien wie *web.xml* und faces-*config.xml* erstellt. Die Maven-Konfiguration wird hierbei um die benötigten Abhängigkeiten erweitert. Auf das eben angesprochene Konzept der Facets gehen wir in einem späteren Abschnitt nochmals ein. Zwar könnte das Webprojekt bereits jetzt in einem Server gestartet und über den Browser aufgerufen werden, jedoch würde lediglich eine fast leere Welcome-HTML-Seite zu sehen sein.

Persistenz konfigurieren

Da ein Webprojekt ohne Persistenzschicht unvollkommen wirkt, konfigurieren wir diese mit dem Befehl *$ persistence setup --provider HIBERNATE --container JBOSS_AS6*. Wir haben uns in diesem Beispiel für Hibernate als Persistence-Provider und die Zielumgebung JBoss AS 6 entschieden. Wie bei allen Kommandooptionen können mittels der Autovervollständigung die zur Verfügung stehenden Möglichkeiten angezeigt werden. So stehen neben Hibernate standardmäßig auch OpenJPA und die JPA-2.0-Referenzimplementierung EclipseLink als Persistence-Provider zur Auswahl. Nachdem die Persistenz konfiguriert ist, benötigen wir noch die zu persistierenden Entitäten. In unserer kleinen Demoanwendung entscheiden wir uns für die Erstellung einer einfachen Entität für einen Kunden mit Attributen für Vorname und Nachname:

```
$ entity --named Customer
Customer.java $ field string --named firstName
Customer.java $ field string --named lastName
```

Nach der Erstellung der Entität befindet sich der Kontext der Shell auf der eben erstellten Entität. Mit dem Befehl *ls* können in der Konsole alle Attribute und Methoden aufgelistet werden. An dieser Stelle kommt erneut das Scaffolding zum Einsatz. Ist der Kontext der Shell auf einer Entität gesetzt, so können mit folgendem Befehl zu der Entität die passenden JSF-Seiten für CRUD-Operationen generiert werden: *Customer.java $ scaffold from-entity*. Für Datenbankzugriffe wird zudem eine Persistenzschicht angelegt. Diese enthält nach der letzten Befehlsaus-

führung eine Java-Beans-Klasse für die Kundenentität, die die üblichen Methoden eines DAOs implementiert. Die Persistenzschicht basiert auf dem Seam-Modul Seam Persistence. Für das Testprojekt haben wir das Persistenzmodul in der Version 3.0.0. Final verwendet.

Projekt bauen und starten

Das erstellte Projekt kann direkt über die Forge Shell gebaut werden. Hierzu muss in der Shell lediglich der Befehl *build* eingegeben werden. Intern wird durch diesen Befehl ein Maven Build angestoßen. Alternativ kann das Projekt auch klassisch direkt mit Maven ohne Forge-Konsole gebaut werden. Die im Build erzeugte *WAR*-Datei ist lauffähig und kann anschließend direkt in den JBoss-Applikationsserver kopiert werden. Nachdem der Applikationsserver mit der Demoanwendung gestartet wurde, ist die Webanwendung unter [2] erreichbar.

Im Rahmen des Beispiels wurde ein Java-EE-6-Webprojekt erzeugt, das in einen Servlet-Container oder Applikationsserver eingesetzt werden kann. Aus dem Java EE 6 Stack wurden die Technologien CDI, JSF 2.0 und JPA 2.0 verwendet. Mit Seam Persistence kommen auch die Standardfunktionen aus dem Seam-3-Framework hinzu. Das Beispielprojekt läuft prinzipiell im JBoss-Applikationsserver 6 und 7 und steht unter [3] zum Ausprobieren bereit – prinzipiell, da das Deployment leider nicht so einfach durchzuführen ist, wie man es eigentlich erwarten würde. Da das aber nicht die einzige problematische Stelle ist, verweisen wir hier auf den Kritikteil am Ende des Artikels.

Konzepte

Bei der Gestaltung der Architektur lag das Augenmerk auf der Möglichkeit, Seam Forge um neue Funktionen einfach erweitern zu können. Forge bietet daher ein generisches Basis-API, das auf verschiedenen Erweiterungskonzepten basiert. Die für die Erweiterbarkeit implementierten Konzepte sind Plug-ins, Facets und Scaffold Types.

Das Plug-in-Konzept bildet das Herzstück des Erweiterungsmechanismus von Forge. Genau genommen besteht das Framework selbst aus einer Sammlung von Plug-ins, hierzu zählt zum Beispiel das bereits vorgestellte Persistence-Plug-in. Sie können recht einfach geschrieben und verwendet werden und stellen somit eine große Bereicherung auf dem Weg zur eigenen Projektkonfiguration dar. Mit den folgenden beiden Befehlen wird das Grundgerüst eines Plug-in-Projekts erstellt:

```
$ new-project --named myplugin --topLevelPackage com.acme
$ project install-facet forge.api
```

Wie bereits vorgestellt, erzeugt der erste Befehl ein neues Projekt. Die Ausführung des zweiten Befehls bewirkt die Installation des Forge API Facets, mit dem in der Projekt-POM die für die Entwicklung eines Plug-ins notwendigen Maven-Abhängigkeiten eingetragen werden. Zum einen werden mit Facets also die benötigten Bibliotheken eingebunden, zum anderen halten Facets auch Kontextinformationen bereit. Facets werden zudem von Plug-ins benötigt, wenn sie Ressourcen erstellen oder ändern. So wird zum Beispiel für das Bearbeiten von Java-Klassen das *JavaSourceFacet* benötigt. Nachdem das Grundgerüst des Plug-in-Projekts erstellt und die benötigten Bibliotheken eingebunden wurden, kann die Plug-in-Klasse erstellt werden. Das erfolgt jedoch nicht über die Forge Shell, sondern händisch in der Entwicklungsumgebung der Wahl. Die erstellte Plug-in-Klasse (Listing 1) muss das Interface *org.jboss.seam.forge.shell.plugins.Plugin* implementieren.

Die auf Klassenebene platzierte Annotation *@RequiresProject* besagt, dass man sich für die Ausführung des Plug-ins in der Forge-Konsole innerhalb eines Projekts befinden muss. Mit der Annotation *@Alias* wird der Name des Plug-ins definiert. Es wird nach einer Installation in der Shell über den Aliasnamen angesprochen.

Plug-ins bieten Zugriff auf die implementierte Funktion über so genannte Commands. Damit eine Methode als Command vom Plug-in zum Aufruf angeboten wird, muss diese entweder mit *@DefaultCommand* oder mit *@Command* annotiert sein. *DefaultCommand* kennzeichnet die Methode, die aufgerufen wird, falls das Plug-in ohne die Angabe des auszuführenden Command-Namens aufgerufen wird. In unserem Beispiel-Plug-in würde der folgende Aufruf dazu führen, dass die Methode *exampleDefaultCommand* aufgerufen wird: *$ myplugin*. Command-Argumente müssen durch die *@Option*-Annotation gekennzeichnet werden. Diese Annotation bietet verschiedene Konfigurationsmöglichkeiten an. So kann ein Command-Argument unter

Listing 1

```
@Alias("myplugin")
@RequiresProject
public class ExamplePlugin implements Plugin {

  @DefaultCommand
  public void exampleDefaultCommand(@Option String opt, PipeOut out) {
    out.println(">> default command: " + opt);
  }

  @Command("perform")
  public void exampleCommand(
      @Option(defaultValue = "DEFAULT", name = "option1")
              String option1,
      @Option(required = true, name = "option2",
   shortName = "o2") Double option2,
      PipeOut out) {
    out.println(">> perform: " + option1 + " and " + option2);
  }
}
```

anderem einen Namen haben, unter dem er direkt in der Shell angesprochen werden kann. Weiterhin kann ein Argument als obligatorisch gekennzeichnet werden, einen Kurznamen und einen Standardwert besitzen. Ein Teil der möglichen Einstellungen ist in der *exampleCommand*-Methode des Beispiel-Plug-ins (Listing 1) enthalten. Ein Aufruf dieser Methode könnte wie folgt aussehen: *$ myplugin perform --option1 A --option2 22*.

Installation von Plug-ins

Abschließend zum Thema Plug-ins werfen wir noch einen Blick auf die Installationsmöglichkeiten. Plug-ins können in der lokalen Forge-Installation über die Forge Shell einfach eingelesen und direkt verwendet werden. Hierzu existieren die folgenden drei Möglichkeiten:

```
$ forge source-plugin   [path\to\plugin]
$ forge git-plugin   [git repo]
$ forge mvn-plugin   [pluginIdentifier]
```

Mit dem ersten Aufruf kann ein eigenes Plug-in direkt vom lokalen Projekt bezogen werden. Der Befehl baut aus den Sourcen das Plug-in und bindet dieses in die Forge-Laufzeitumgebung ein. Der zweite Aufruf verlangt die Angabe eines Git Repositories, während beim dritten das bereits gebaute Plug-in anhand des Identifiers aus einem Maven Repository geladen wird.

Abschließend zu den Konzepten werden wir auch noch ein paar Worte zu einem ebenfalls sehr wichtigen Konzept verlieren: dem Scaffolding. Es besteht aus zwei Teilen. Ein Teil betrifft das Generieren des Grundgerüsts der Webanwendung inklusive der Erzeugung von CRUD-Formularen. Ein anderer wichtiger Aspekt ist die Technologiewahl für die erzeugten Formulare. Seam Forge setzt hier in der aktuellen Version zunächst nur auf JSF, bietet aber per API die Möglichkeit, in Zukunft auch weitere Frontend-Technologien in die Generierung einzubinden. Die Grundlage dafür bilden die so genannten Scaffold Types. Mit dem einzigen bisher vorhandenen Scaffold Type sind wir bereits bei der Ausführung des Befehls *scaffold from-entity* in Kontakt gekommen. Es handelt sich hierbei um den Scaffold Type für JSF. Die Implementierung eines eigenen Scaffold Types folgt dem Provider-Pattern. So reicht es aus, in einem Forge-Plug-in eine Implementierung der Schnittstelle *ScaffoldProvider* bereitzustellen. Ein Beispiel einer konkreten Implementierung liefert der Standard-Scaffold-Type *MetawidgetScaffold*. Nach der Vorstellung der Basisfunktion und einer Beschreibung der zugrunde liegenden Konzepte muss jedoch als Nächstes auch Negatives berichtet werden.

Kritik

Die Idee einer einfachen, leicht konfigurierbaren, aber auch generischen Projektinfrastruktur klingt an sich sehr verlockend. Seam Forge bringt konzeptionell auch vieles mit, um diese Herausforderung zu meistern. Im praktischen Einsatz haben sich jedoch nicht wenige Schwächen offenbart, von denen wir im Folgenden die wichtigsten nennen:

- Sowohl die Projekt- als auch die Codedokumentation ist unzureichend. Viele Konzepte sowie das API werden in der Dokumentation sehr oberflächlich beschrieben. Teilweise gibt es zu bestimmten Aspekten gar keine Dokumentation (z. B. zu Facets oder zur Implementierung eigener Scaffold Types). Das erfordert das mühselige Wühlen im ebenfalls so gut wie gar nicht dokumentierten Sourcecode.
- Ein Deployment gegen den JBoss AS 7 funktioniert nicht aufgrund einer Kombination aus Metawidget und AS 7 Classloadern. Unter JBoss AS 6 funktioniert die aktuelle Version Beta 1 aufgrund eines Bugs nur unter Verwendung der Java-Version 1.6_24 oder älter [4].
- Einige der Standard-Plug-ins sind entweder fragil oder können zunächst nicht benutzt werden. So gibt es zum Beispiel auch ein Plug-in, mit dem einfache Java-Klassen erstellt werden können. Aufrufe des hierfür angebotenen Plug-ins scheitern mit einer nichtssagenden Fehlermeldung. Erst nach einem Blick in den Quellcode wird klar, wie es eingesetzt wird. Hierzu ein Beispielaufruf des sich sehr unnatürlich anfühlenden Plug-ins: *$ java new-class --package com.acme public class MyClass{}*.
- Es gibt viele fehlende Feinabstimmungen auf unterster API-Ebene. So wird eine konfigurierte Klasse bei jeder auf Forge basierten Änderung komplett neu formatiert. Das wird im Projektalltag schnell zur Last.

Wie geht es weiter?

Das nächste Release 1.0.0. Beta ist für September geplant. Ein finales Release könnte Ende des Jahres folgen. Neben weiterer Stabilisierung des API und der Containeranbindungen sind einige neue Features geplant. Hierzu zählt zum Beispiel eine erweiterte Navigation für CRUD-Masken und die Stabilisierung des zentralen Plug-in-Repositories. Da sich Plug-ins in Zukunft als Zugpferd des Frameworks erweisen könnten, könnte das Projekt durch ein zentrales Repository endlich an Fahrt zunehmen.

Fazit

Unser Fazit auf die Titelfrage „From Zero to Hero?“ lautet momentan leider eindeutig: Weder Zero noch Hero. Auch wenn die Konzepte überzeugen mögen, scheint Forge noch nicht ausgereift und reicht daher momentan lediglich für Prototyping. Die angestrebte Vielseitigkeit klingt verlockend, birgt jedoch auch die Gefahr, den Fokus zu verlieren. Dieser könnte jedoch wichtig sein, um eine Community in entsprechendem Umfang für sich zu gewinnen. Als sehr störend empfinden wir den fragilen Zustand des API und der Standard-Plug-ins. Viele der Befehle funktionieren nicht auf Anhieb oder sind unvollständig implementiert. Das ist insofern problematisch, da man teilweise zwischen der Shell und einem Editor hin und her schalten muss. Damit Änderungen in der Konsole verfügbar sind, muss diese sogar in bestimmten Fällen neu gestartet werden. Zum aktuellen Zeitpunkt wäre unserer Ansicht nach ein stärkerer Fokus auf die Stabilisierung des API und eine gründlichere Dokumentation sinnvoller als die Umsetzung neuer Features. Dennoch wurde durch die eben vorgestellten Konzepte eine viel versprechende Grundlage geschaffen, mit der Forge um beliebige Erweiterungen bereichert werden kann. Man darf gespannt sein, in welche Richtung sich das Framework durch die derzeit entstehende Community in Zukunft entwickeln wird.

Michael Schütz arbeitet als Senior Software Engineer bei der akquinet AG in Berlin und beschäftigt sich seit Jahren mit Java-Enterprise-Architekturen und -Technologien. Er ist Gründer und Moderator der Seam-Plattform auf Xing und wirkt regelmäßig an JBoss-Open-Source-Projekten mit. Michaels berufliche Interessen liegen derzeit in den Bereichen Seam/CDI, Enterprise Portal, Java EE Testing und Git. Kontakt: Michael.Schuetz@akquinet.de.

Marek Iwaszkiewicz ist für die akquinet AG als Software Engineer tätig. Den Schwerpunkt seiner Arbeit bilden Java-EE-Architekturen und -Technologien. Sein aktueller Fokus liegt in den Bereichen Seam und jBPM. Kontakt: Marek.Iwaszkiewicz@akquinet.de.

Links & Literatur

[1] http://metawidget.org

[2] http://localhost:8080/testprojekt/faces/scaffold/customer/view.xhtml

[3] https://github.com/michaelschuetz/seam-forge-sandbox

[4] https://issues.jboss.org/browse/WELD-897

[5] http://seamframework.org/Documentation/SeamForge

[6] Forge-Referenzdokumentation: https://docs.jboss.org/author/display/SEAMFORGE/Reference+Guide

[7] Forge-Videos: http://vimeo.com/channels/seamforge

[8] Forge-Quellcode: https://github.com/forge

[9] http://www.theserverside.com/discussions/thread.tss?thread_id=62154

[10] Democode: https://github.com/michaelschuetz/seam-forge-sandbox

[11] Eigenes Beispiel-Plug-in: https://github.com/michaelschuetz/forge-log-plugin

[12] Metawidget Scaffold: http://bit.ly/pKUwy7

[13] http://it-republik.de/jaxenter/artikel/Metawidget-King-of-the-Hill-2681.html

[14] http://metawidget.sourceforge.net/documentation.php